Separable User Interface Architectures in Teaching Object Technology

Rob Griffiths, Simon Holland, Mark Woodman , Malcolm MacGregor and Hugh Robinson

Computing Department
The Open University
Walton Hall
Milton Keynes, England MK7 6AA
+44 1908 274066
r.w.griffiths, s.holland, m.woodman, m.d.macgregor, h.m. robinson@open.ac.uk

This paper appeared as:

Griffiths,R., Holland, S., Woodman, M., MacGregor, M. and Robinson H. (1999) Separable User Interface Architectures in Teaching Object Technology. Pages 290-299 Proceedings of the IEEE Thirtieth International Conference on the Technology of Object Oriented Languages and Systems, TOOLS USA '99, Santa Barbara, USA August 1999. ISBN 0-7695-0278-4.

ABSTRACT

This paper concerns the critical role of separable user interface design in teaching object-oriented systems. M206 "Computing: An Object-oriented Approach" is a large-scale university-level introduction to software development designed from scratch for distance learning, using an objects-first approach with Smalltalk. The course is degree-level, counting as one sixth, and is being offered in the UK, Western Europe and Singapore. To address the needs of industry we have developed a radical syllabus that adheres to the principle of designing complex systems by separating view and model, and have developed a programming and learning environment to support these ideas. In the paper we examine how separable user interface architectures have guided our teaching of object technology and the design of powerful microworlds that are both usable and extendible by neophytes. The course and relevant teaching with software is outlined and the technical design and pedagogic use of the microworlds and GUI builder tool are described.

Separable UI Architectures in Teaching Object Technology

Rob Griffiths, Simon Holland, Mark Woodman, Malcolm Macgregor, Hugh Robinson
Computing Department, The Open University, Walton Hall,
Milton Keynes, England MK7 6AA
tel: +44 1908 274066
email: r.w.griffiths, s.holland, m.woodman, m.d.macgregor, h.m.robinson@open.ac.uk

Abstract

This paper concerns the critical role of separable user interface design in teaching object-oriented systems. M206 "Computing: An Object-oriented Approach" is a large-scale university-level introduction to software development designed from scratch for distance learning, using an objects-first approach with Smalltalk. The course is degree-level, counting as one sixth, and is being offered in the UK, Western Europe and Singapore. To address the needs of industry we have developed a radical syllabus that adheres to the principle of designing complex systems by separating view and model, and have developed a programming and learning environment to support these ideas. In the paper we examine how separable user interface architectures have guided our teaching of object technology and the design of powerful microworlds that are both usable and extendible by neophytes. The course and relevant teaching with software is outlined and the technical design and pedagogic use of the microworlds and GUI builder tool are described .

INTRODUCTION

The Open University (OU) is the UK's largest university; its primary mission is to make higher education available to adults regardless of their personal circumstances and earlier educational achievements and its courses are offered almost exclusively in the distance mode. Since it was established in 1969 more than two million people have studied with the OU the UK, Europe and world-wide. After four years of development, the OU has launched its flagship course *Computing: An Object-oriented Approach* – a radical introduction to designing and writing complex software systems. The title is intended to convey nuances about the course not readily discernible from *Computer Science*, or *Computer Studies*, or *Software Engineering*, or *Informatics* and emphasises our commitment to Object Technology and its general applicability. However the course is not just about object-oriented programming, it includes analysis and design loosely centred around the CRC approach of Wirfs-Brock *et al*. [1] but with a flavour of the more formal treatment of associations given by Cook and Daniels [2] and Human–Computer Interaction, especially the separation of domain model and user interface classes which we make central to our pedagogy. Syllabus and multimedia details are on the Web (at http://www-cs.open.ac.uk/~m206/). The main course objectives were that, after successfully studying the course, students should:

- have a general understanding of computing and software, including an extensive vocabulary;
- have obtained practical, generally applicable skills in using software, e.g. programming, telecommunications and multimedia software;
- have sufficient knowledge of the object-oriented paradigm to analyse software artefacts and problems in terms of it, to sketch the design of systems, and to make design choices, and to complete or extend an application;
- be capable of developing small applications including appropriate graphical user interfaces.

After much debate [3] we decided on an 'objects-first' approach to computing and to adopt a pure object-oriented language for the course, thereby avoiding the confusion and misconceptions that can arise when teaching a hybrid language. This was especially important as the course is designed for the distance learning mode where individual practical help is not immediately at hand. The programming language which best suited our agenda was Smalltalk-80 [4, 5], an industrial-strength language and arguably the first really practical object-oriented programming language. For us its primary benefits were that the language is based on just a few concepts and its programming environments have the following properties:

- they are simple embodiments of object technology;
- they lend themselves to tailoring;
- all aspects of the system can be explored interactively and reflectively

- they are suitable environments in which to produce student-alterable microworlds as they allow a clean separation between model and user interface.

The curriculum is designed for students planning to major in computing, and also to be fully comprehensible to students planning to major in other subjects. This particular requirement, coupled with the wide ability-range of mature distance students (average age 37), posed a number of problems. In order to address these problems, we developed a new programming environment that integrates the following elements: a purpose-designed variant of the MVC graphical user interface architecture, a specially designed GUI builder, a set of custom built programming tools, and a series of educational microworlds. Our programming environment is designed to progressively reveal its more powerful and complex features as the course unfolds. It initially appears to be extremely simple, but by the end of the course it is comparable in power to an industrial-strength programming environment, which enables our students to develop non-trivial applications with fully separable graphical user interfaces, i.e. applications where model and view are designed separately and are entirely modular. Moreover, the introductory microworlds are designed with similar properties of progressive disclosure: unlike many other simulations, the microworlds progressively allow students to control , modify (e.g. add new behaviour or state), and extend them.

For those in our degree programme the course is worth one sixth of a degree and goes beyond what any organisation has attempted by providing to ordinary *users* of computer systems the resources to become *developers* of them. The course introduces computing to thousands of people per year and radically recasts it for others. Currently M206 is enrolling over 5,000 students per year in the UK and an additional 600 are taking the course at the Singapore Institute of Management. In late 1999, the course will become available in the USA. The social and educational impact of tens of thousands first learning to think of software in object-oriented terms will be both dramatic and immediate.

In this paper we examine how separable user interface architectures enable the design of powerful microworlds that are both usable and extendible by neophytes. The background for this work -- the course M206 and our programming environment - is outlined, and the design and pedagogic use of the microworlds and GUI builder tool are described.

LEARNINGWORKS

Having concluded that object-oriented technology and Smalltalk were appropriate vehicles for our teaching, the team considered the state of available Smalltalk environments. The requirement that the course be accessible to non computer science majors led us to the conclusion that the available environments were too complicated for real beginners. We needed an environment that would make the experience of beginning students as worry-free as possible. For example:

- Some form of simple switching between applications or text windows to attend to the difficulties of window navigation for people with little sophistication in this area.
- A range of parameterised browsers to avoid the paralysis felt by many people when confronted by a dauntingly large class library and who mistakenly think they need to understand it all before proceeding.
- Simple and helpful evaluation and inspection facilities that would improve upon the ways in which Smalltalk environments provide transcript and workspace windows.

We concluded that we would have to develop our own environment. However, shortly after embarking on our own development we established a collaboration with the Smalltalk pioneer Adele Goldberg and her colleagues on an object-oriented framework called LearningWorks [6]. LearningWorks allows a Smalltalk environment to be used as a vehicle for interactive courseware and programmers to specialise it to produce their own customised system. As far as learners are concerned LearningWorks offers a set of “LearningBooks”, which are software modules that are presented to the user as a book metaphor: a LearningBook is organised into sections, and sections into pages. Each page being an arbitrary application. And, like in a loose-leaf binder, a page may be “detached” from its book and left conveniently on the desktop, allowing the user to continue to view a page from one section, having moved to another. LearningWorks, also provides a class and method scoping mechanism for LearningBooks called ‘The Vision’.

This framework proved ideal for our pedagogy, as we were able to write our own microworlds and programming tools as pages for LearningBooks, furthermore the scoping mechanism enabled our browsers to progressively reveal more of the class library as the student worked through the

LearningBooks. So initially we can present our students with LearningBooks that appear to be extremely simple programming environments, where very few classes are made visible, and within these very little visible implementation code. By the end of the course however we can present LearningBooks which are comparable in power to industrial-strength programming environments[7]. Typical LearningBook pages are shown in Figure 1.

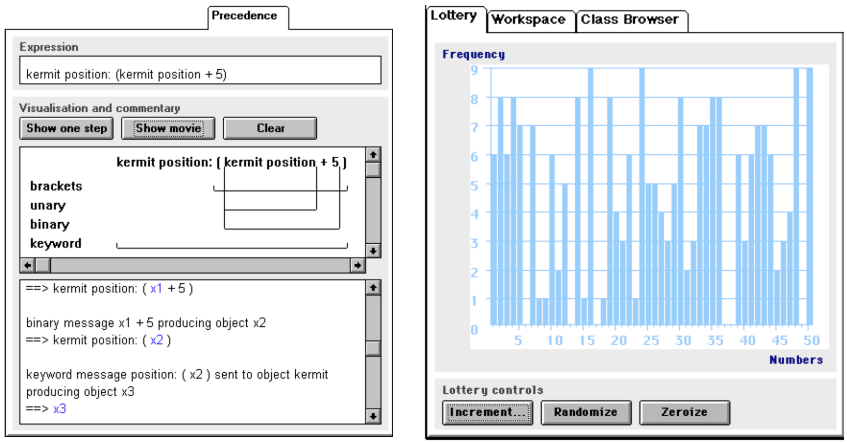


Figure 1

SEPARABLE USER INTERFACES

Smalltalk is inextricably linked with simulation [8]. Indeed, an object-oriented system is often described as a simulation, or model, of some part of the real world or a business enterprise. We were therefore culturally well disposed to introduce simulations into our interactive learning environment in the form of microworlds. One in particular was used as a touchstone for reasoning about object concepts – an amphibian microworld that models the behaviour of instances of classes `Frog` and `Toad` and of a subclass of `Frog`, `HoverFrog`. The simulation has been devised to expose all object concepts, starting with the simplest: initially the simulation shows objects of the classes `Frog` and `Toad` which a have identical state attributes – `position` and `colour` – and identical message protocols, such as `green`, `brown`, `home`, `right` and `left`, which respectively set the receiving object's colour to green, and brown, and change its position to the 'home' one and move left and right. Students select any of the objects in the microworld from a regular scrolling list (which they much later discover is another object!) and use the GUI widgets to send the corresponding messages. This simple user interface not only allows straightforward message sending to be visualised, it allows apparently advanced notions such as polymorphism to be demonstrated; for example, when a frog is selected and the **home** button is clicked (resulting in the message `home` being sent) the receiving frog moves to the leftmost position, but if a toad has been selected, and so receives the message `home`, it moves to the *rightmost* position – the 'home' position for toads.

Right from the beginning of the course we introduce our students to the notion of domain models and separable user interfaces. An innovation of LearningWorks we used heavily to sustain this idea is the provision of *page-local* and *section-local* variables, implemented in page and section dictionaries respectively. In practice the latter are the most useful. For example, any objects of any class created in a Workspace page and assigned to section-local variables are accessible by all pages in that same section. Therefore, to facilitate the visualisation of separable user interfaces, assignment, object creation and disposal, our amphibian microworld was built not as an arbitrary application, but, in effect, as a specialised graphical view of the section variable dictionary. This graphical view of the section dictionary is specialised in the sense that it only displays objects of certain classes of initial interest (e.g. frogs, toads, hoverfrogs). The result of this design decision is that simply creating an object of the relevant class in the Workspace page and assigning it to a section variable will cause its graphical representation to appear in the amphibian microworld page automatically.

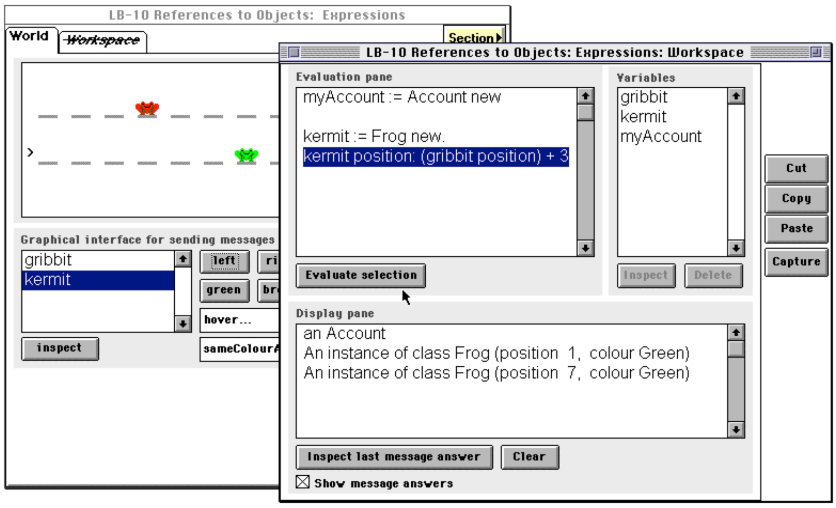


Figure 2

Figure 2 shows a LearningBook which is open at the Amphibian World page. Overlaid, next to it is a Workspace page that has been 'detached' from the same section of the book, i.e. placed on the desktop. The scrollable list to the top, right of the workspace is a list of local variables: `gribbit`, `kermit` and `myAccount`. The variable `myAccount` is currently referencing an `Account` object and therefore the Amphibian World will not display it, however the microworld is displaying the `Frog` objects referenced by `kermit` and `gribbit`. Students can send messages to these amphibian objects either by clicking buttons on the Amphibian World page, or by textually sending messages to amphibian objects in the Workspace page. In either case any state-changing messages such as `right`, `left` or `colour:` will be reflected in the Amphibian World.

This combination of microworld and Workspace provides a powerful visual aid in the teaching of reference semantics and assignment and in particular helps our students avoid a number of misconceptions [9] such as:

- only one variable can reference to a given object at a given time;
- once a variable references a given object, it will always reference that object;
- a variable that refers to an object uniquely specifies it for all time;
- if you have two different variables, they must refer to two different objects.

For example if a student evaluates the assignment expression `gribbit := kermit`, the `Frog` object previously referenced by `gribbit` will have no remaining references to it, and the automatic garbage collection of that object will be graphically dramatised in its immediate disappearance from the microworld. The fact that `gribbit` and `kermit` now reference the same object can also be graphically demonstrated as state changing messages to the sole remaining frog object can be sent via `kermit` or `gribbit`. Students quickly become proficient in this style of microworld manipulation and programming and our account of how user interfaces are implemented and separated from domain models is a crucial part of our pedagogy. Furthermore, due to the separable interface discipline, all of the relevant model code can be inspected, modified and understood by students without them being distracted by the interface code. Eventually they are able to construct new kinds of objects from the ground up and – provided they are subclassed from a displayable object and no new state added – the objects, their state and their behaviours will all automatically be displayed in the microworlds, without students having to pay any attention to explicit graphical interface programming. For example students implement a subclass of `Frog` called `Siren`, instances of which move right or left by `0.1` of a position; have an initial colour of `blue`, and have position `2.7` as their home position.

As the course progresses students are introduced to other microworlds, for example a microworld simulating of an air traffic control system.

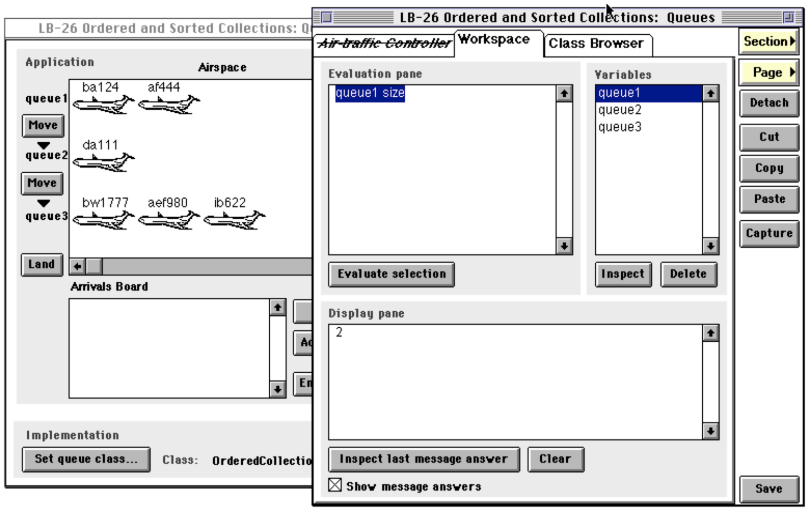


Figure 3

Figure 3 shows two pages: an air traffic control microworld (detached from the book) and a Workspace, where issues of design and safety-critical software development are explored. The microworld is a graphical representation of three queues of plane objects. In our teaching we discuss the necessary behaviour for these queues and discussion leads students to suppose that a subset of the `OrderedCollection` protocol would provide the necessary behaviour. Hence initially the queues are instantiated as instances of `OrderedCollection`. The queues are referenced by section local instance variables so students can send messages to them from the user interface or from the Workspace. When these queues are manipulated from the microworld, planes can be landed safely as the GUI only sends the agreed subset of the `OrderedCollection` protocol to the queues. However from the Workspace students are free to send any of the messages from the `OrderedCollection` protocol to the queues, sometimes with dramatic results. For example sending `at:put:` messages to a queue will cause planes to be lost from the system. Once they realise how easily the queue abstraction can be broken we can teach them various strategies for designing and implementing a bespoke `Queue` class. Once they have implemented this new class they can trivially instantiate the queues in the microworld as instances of this new class.

GOALS FOR GUI IMPLEMENTATION

As outlined above, our pedagogy very clearly establishes the separation of domain model and user interface. Given this, for the notion of separable user interface architectures to mean anything to novices, they have to be able make concrete the abstract by eventually implementing their own user interfaces using a GUI builder, and to write all of the domain model and user interface code necessary to make these interfaces work. The most general goal of this part of our pedagogy was that students should understand, in detail, the advantages of allowing user interface and domain model development to be pursued independently. Specifically, our students should be able to:

- create their own user interfaces, and connect them to domain models, thus creating applications.
- understand and use the broadcast dependency mechanism and the messages involved;
- alter existing user interfaces built with a GUI builder;
- give a model multiple user interfaces simultaneously;
- reconnect a user interface to different domain models (i.e. to a different instance, or, to an instance of a different class where appropriate).

One of the earliest and most developed architectures in Smalltalk is Model/View/Controller (MVC) [10]. MVC is an approach to application development that divides the application into the information of interest (model), the visual representation of that information (view), and the handling of user input (controller). This was the first widely used architecture for user interfaces that allowed models (i.e. application code) to be written quite independently of their user interfaces, and which allowed user interfaces to be modified, replaced or run several at the same time, without the need for any changes at all to the model. The model need not know whether it has a user interface, or how many user interfaces it has. These characteristics are the hallmark of a separable user interface.

In order to allow various kinds of flexibility, VisualWorks instantiation of MVC introduces a kind of impedance matcher or buffer class whose instances stand between the model and the various widgets. This class is called `ApplicationModel`. In effect, application models absorb the messiness of practical linking between model and user interface widgets, and translate messages from widgets in the `value`/`value:` protocol into the domain specific protocol of the model in question. The cost of this approach is that a lot of knowledge about both domain model and user interface needs to be built into the application model by the programmer. Indeed, the application model may end up mirroring and hence duplicating much of the domain model. This architecture works very well for professional programmers. It allows tremendous flexibility in the fine detail of how widgets can be used, and it keeps user interface logic out of domain code. However, even for experienced programmers, using the MVC mechanism [11] can be very daunting. For any given application there are will be multiple models, views and controllers. For example, the 'model' for even a small application will consist of the application model, and possibly multiple models from the domain model. The user interface will consist of many views (the graphical representation of widgets). Each widget will be a dependent of some different aspect of the application model rather than the whole user interface simply being a dependent of some domain object. Therefore, for the beginner, this complexity tends to obscure the essential simplicity of the separable user interface idea.

In a degree course aimed principally at first-time computing students, the complexity of the MVC architecture behind the VisualWorks GUI builder poses a problem that is not mitigated by the friendly and simplifying front end of LearningWorks, even with our principle of progressive disclosure [7]. We did not want students merely to learn to use a GUI builder. We wanted them to understand and to work with separable user interfaces, to understand the architecture that makes them possible. To this end we devised a simplified version of MVC which we term MUI, Model–User Interface. MUI, is an architecture that allows beginners to easily create GUI applications by binding domain objects (models) to instances of user interface classes that they create using a simple GUI building tool. At each stage of the process extensive checks are carried out and the runtime environment ensures that problems do not result in cryptic Smalltalk exceptions but instead are reported in an understandable way (in the vocabulary of the course), after which a safe recovery is made.

Before we introduce the MUI architecture to our students, they are taught about the broadcast dependency mechanism from which the *Observer* pattern is abstracted [12, 13]. Using this mechanism one object (the observer) is made a dependent of another object (the observed). They learn that to notify an observer object of any state changes the observed must include `self changed` message expressions in its state-changing methods. The model's only responsibility is to notify its dependents when its state changes. In that way the model need not take any account of what its dependents are, or indeed whether it has any. The responsibility is on any user interface components to respond appropriately to such notifications, and, where appropriate, to query the model about its current state so that they can update themselves suitably.

Our design goal for MUI was that an arbitrary model could be bound to an arbitrary user interface via this broadcast dependency mechanism. In other words, we wanted an architecture in which a user interface class could be made a direct dependent of a model rather than the individual widgets being dependent on value models in an application model as is the case with the VisualWorks instantiation of MVC. Furthermore, as long as the model provided the protocol expected by the widgets in the user interface it should simply work without any further 'glue' code being written by the user. In fact, the only user interface related code that appears in the model is the `self changed` expression mentioned above so that the user interface can be alerted via the broadcast dependency mechanism that the state of the model has changed. There are three basic components to the architecture which we subsequently outline: the OpenGUI tool, the user interface widgets and the test page.

This OpenGUI tool appears as a page in a LearningBook (see Figure 4). It looks like a fairly standard (if very simple) tool for laying out interface widgets and giving them property values. (This tool is subclassed from the drawing application used earlier in the course, so its user interface is familiar to students.) OpenGUI supports the following widgets: label, divider, group box, action button, check box, radio button, slider, text editor, input field and list box. We considered various ways of implementing menu buttons and other more complex widgets, but inevitably interface code ends up in the model and so they were rejected.

The amphibian microworld described earlier displays graphical representations of `Frog` objects. Their protocol includes the messages `position` and `position:`, which are used to get and set the

`position` instance variable representing a frog's position attribute. With this information students can construct an alternative user interface to control instances of the `Frog` class.

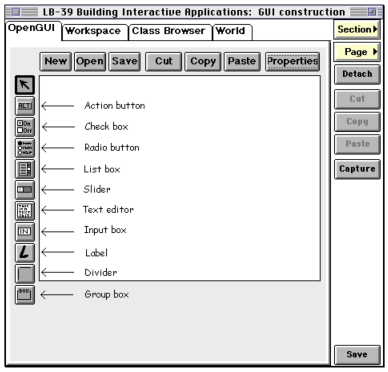


Figure 4

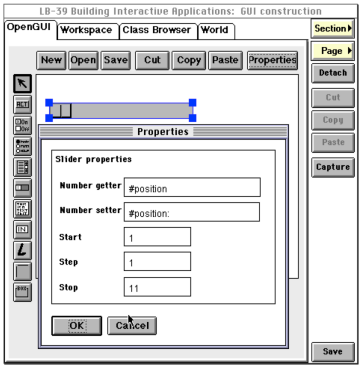


Figure 5

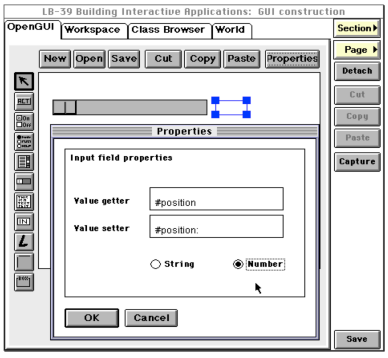


Figure 6

In the OpenGUI page the student selects the slider drawing tool and draws the slider to the required size on the canvas. With the drawn slider still selected the user clicks on the **Properties** button to open a dialogue box which will allow the properties of the slider to be set. (See Figure 5.) As sliders can only work with numerical information the user is asked to supply the names of messages that will get and set a numerical instance variable in the prospective model (in this case a frog). Note that in the VisualWorks GUI builder the user would be asked to supply the name of a value model in the application model, rather than a message from the domain model's protocol. Other values requested are highest and lowest values that the slider can set in the model using the setter message, and the size of the increments the slider can make. To complement the slider an input box can be added to the canvas as in Figure 6.

Note again that the user supplies getter and setter messages from the model's protocol rather than the name of a value model in the application model. As we mentioned earlier some of our widgets are much simpler than the equivalent VisualWorks widgets. As you can see in Figure 5 the OpenGUI input field supports only two types – number and string. Also input fields are always editable. To make 'editable' another property would be trivial to implement but this was resisted in the cause of simplicity. Because we are teaching in the distance mode we have incorporated extensive error checking into both OpenGUI and the MUI mechanism.

When saving the properties of a widget, OpenGUI ensures:

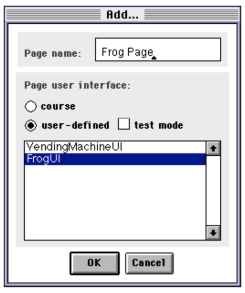
- that none of the fields in the properties dialogue box are blank;
- that any getter is a legal unary selector;
- that any setter is a legal single keyword selector.

Before saving the user interface as a class, OpenGUI checks that:

- the user interface is complete and consistent and the user is warned if they are using the same selector in more than one way in the same user interface;
- the class name chosen, is either a new class name or the name of an existing student-authored user interface class.

The user interface class is then saved as a subclass of `OUGUIAbstractInterface`, itself a subclass of `ApplicationModel` in VisualWorks. Such a user interface class consists of a single class method, `windowSpec`, which specifies how the user interface is to be drawn. When an instance of the user interface class is opened, methods in the superclass `OUGUIAbstractInterface` (which hides the detail of implementation) dynamically create all the value models needed to support the MVC architecture that underpins MUI.

Attaching an instance of this new interface to a suitable model is achieved through the LearningBook's **Page** menu button. Selecting the **Add...** option opens up scrollable list of available user interface classes (Figure 7).



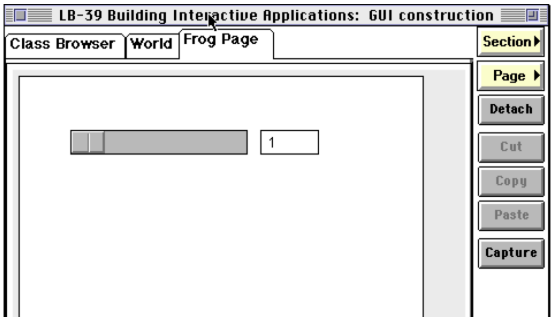


Figure 7 **Figure 8** **Figure 9**

After selecting the desired user interface class the neophyte programmer is then prompted to select an appropriate model from the section dictionary (Figure 8), in other words simply to select a section-local variable. This simple reference to a suitable model must have previously been established in the workspace in the same section.

To guarantee that the contract between user interface and model has been properly established, the MUI behaviour inherited by the particular user interface then carries out extensive checks on the model's protocol and degrades gracefully (see Figure 12) if the right methods are not present in the model, or if they return a message reply of the wrong type. If the checks establish that a contract between user interface and the model can be properly established, an instance of the user interface is created and inserted as the current page in the LearningBook (Figure 9).

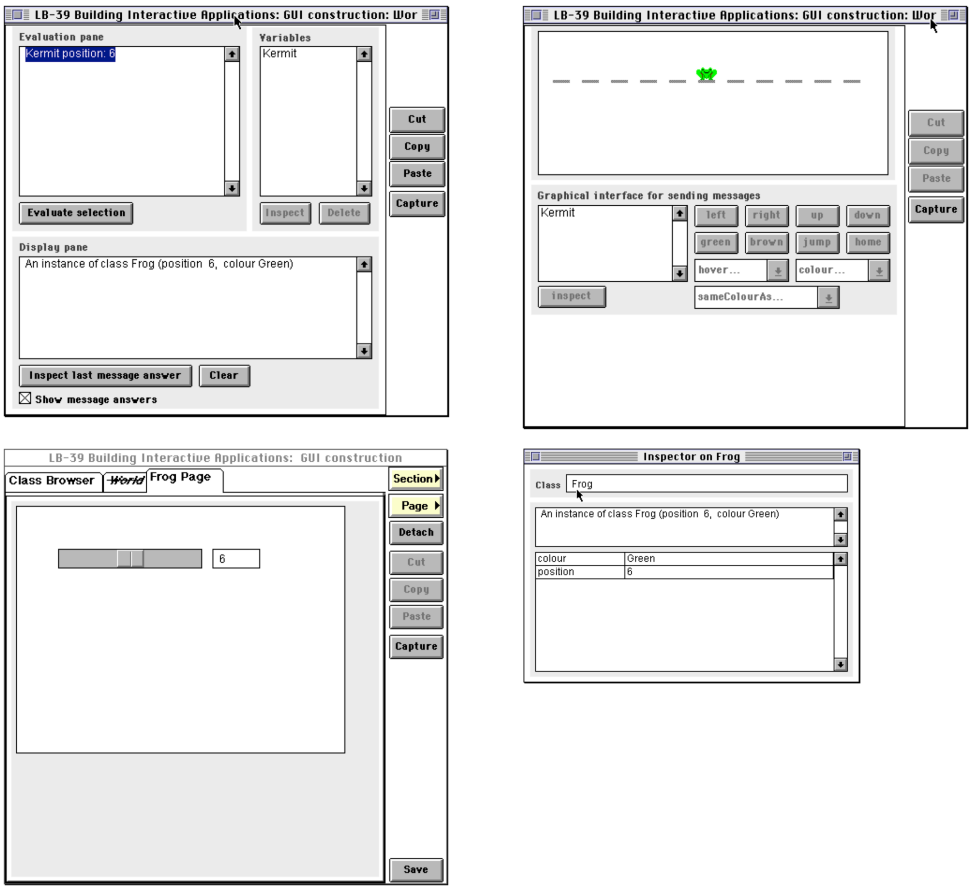


Figure 10

A model from the section dictionary can be associated with any number of user interfaces pages within the same section as the model. Therefore, by 'detaching' a Workspace and an amphibian World from the same section of the LearningBook, and placing them side by side with the new user interface page, students can view the effect of sending state-changing messages to the model from a Workspace, the

state changes will of course be reflected in both user interfaces. Similarly, changing the state of the model from either user interface will be reflected in the other user interface and can be confirmed by inspecting the model from the Workspace. Thus reinforcing the notion of separable user interfaces. See Figure 10.

At any time the user can choose **Test mode** from the **Page** menu button to associate a new model (from the section dictionary) with the page's user interface (see Figure 11). This new model need not be of the same class as the previous model, the only proviso is that it understands the required protocol and that the appropriate setter methods include the `self changed` message expression.

Exceptions and errors involving user interface classes and models in VisualWorks are next to impossible for the neophyte programmer to debug or reason about because of the huge number of complex classes involved. With the MUI architecture, students do not need to see MUI-specific classes or VisualWorks runtime GUI code to understand any errors associated with binding an instance of a user interface class with a model – problems are caught and reported in terms of objects and code created directly by the student. For example, when adding an instance of a user interface class as a page in the LearningBook, if the user interface discovers that the selected model cannot respond to the entire protocol needed by the interface, or if a getter method returns an object of the wrong class, details are reported and no attempt is made to open the interface, instead a test page is inserted into the LearningBook from where the user can select other pairs of local variables and interface classes. In Figure 12, the user has attempted to bind an instance of the `FrogUI` class to a model which is of class Account rather than of class `Frog` – `Account` objects do not understand the message `position:`.

Similarly, once a user interface is opened, MUI detects the following errors:

- that the get or set methods it needs to use are no longer understood by the model (i.e. the user has deleted the method since the interface was opened);
- the return value of a getter or the argument of a setter does not match the appropriate type (because the user has edited the method since the user interface was opened);
- that a method in the model causes an exception.

These errors are reported to the user and a test page is substituted for the current user interface page. Finally, if the method called by a widget results in an infinite loop and the user presses break, they are told which method was probably in the infinite loop and a safe recovery is made.

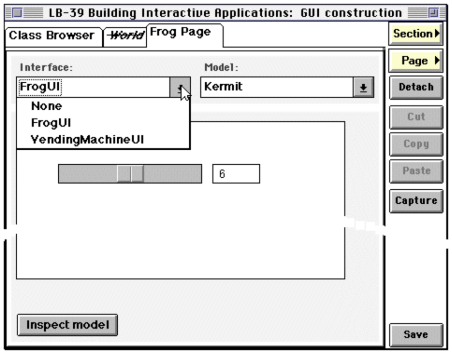


Figure 11

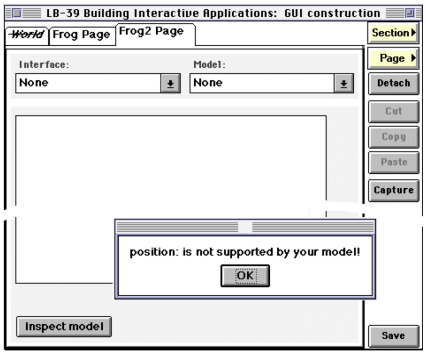


Figure 12

There are some significant restrictions in the current implementation of OpenGUI. For instance, the requirement that a local variable be used to link a user interface to a model means that a collection of models cannot be linked to collection of user interface instances. And, of course OpenGUI is much simpler than a commercial GUI builder, in that it supports fewer widgets, and it assumes that a user interface can only deal with one model at a time (i.e. the views in the user interface only show information from one model). However, it is conceptually straightforward and simple for beginners to use. In effect, we have traded off some loss of flexibility with a tool that allows the unconfident to experiment concretely with all of the key concepts of separable interface architectures.

CONCLUSION

There is an emerging evidence that teaching pure objects first to beginners is a strategy that can provide a conceptually simple and relatively easily understandable foundation for computing, tending to lead to good habits of modularity and fostering a systems-centred view [5]. However, industrial-

strength pure-object programming environments can be intimidating to the complete beginner, particularly in a distance education context or in other situations where individual practical help is not immediately always at hand. We have described how we have addressed this problem by designing, implementing, and delivering to students a programming environment that provides progressive disclosure in a carefully tailored way. This allows, for example, given classes and methods to be hidden initially, and unsafe message expressions to be carefully filtered in the early stages, but with the system ultimately unfolding to a fully permissive industrial strength environment. Another systemic problem we have addressed relates to graphical simulations. Graphical simulations are a good vehicle for promoting the approach outlined above. However, there have historically been practical problems in allowing beginning students to explore, modify, and create from scratch their own graphical simulations in Smalltalk. Firstly, the standard separable interface architecture known as MVC is too complicated for beginning students to understand in their early stages of learning. One historical response to this problem has been to forfeit the principle of keeping this user interface separate from the model, and to allow user interface code to mix with the model code. In exchange for short term simplicity, this introduces longer term problems such as loss of modularity, loss of flexibility, unnecessary coupling and bad design and coding habits. Another possible approach is to provide students with an automated GUI builder, but to pass over in silence how this is linked to the model, and to leave the nature of the connection between the visible parts of the simulation and the underlying model perceived as essentially magical or speculative. This fails to address a central and vital aspect of the simulation approach. We have described in detail our solution to this impasse. Namely, we have designed, implemented, and delivered to students a highly error-protected, dynamic and simplified variant of MVC called MUI that illustrates all of the key principles of separable user interfaces, yet is simple enough for beginning students to understand in great detail. This allows students to explore and modify existing simulations and their user interfaces, and to create their own graphical simulations, applying complete modularity to user interface and model. This is a key component of a curriculum that lays a firm software development foundation for students planning to major in computing, but is also fully comprehensible to students planning to major in other subjects.

More generally, we have deployed a wide range of technologies to support learners grappling with fundamental concepts of object technology in a way that is appropriate to the distance mode and, in particular, provides a transformation from novice to accomplished practitioner.. As we have outlined, we have devised a simple, consistent pedagogy and constructed a programming and learning environment to match the pedagogy, using LearningBooks to package work, systems and tools. We have shown how, using LearningWorks and a wide variety of microworlds and programming tools, novices can progress from using and extending systems through ‘game play’ microworlds, to programming, through object-oriented analysis and design to graphical user interface design and implementation.

References

[1] Wirfs-Brock, R., Wilkerson, B. and Wiener, L. *Designing Object-oriented Software,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] Cook, S. and Daniels, J., *Designing Object Systems,* Prentice Hall Int., Hemel Hempstead, 1994.

[3] Woodman, M., Holland S. and Price, B., Pervasiveness of a Programming Paradigm: Questions Concerning an Object-oriented Approach, *Proceedings CS Education,* Dublin, 1994.

[4] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Reading, MA, 1983.

[5] Woodman, M. and Griffiths, R., Programming Language Choice for Distance Computing, in M. Woodman, *Programming Language Choice,* International Thomson Computer Press, London, 1996.

[6] Goldberg, A., Abell, S., and Leibs, D., The LearningWorks Delivery and Development Framework, *Communications of the ACM*, 40(10), 78–81, 1997.

[7] Woodman, M., Griffiths, R., Macgregor, M., Holland, S., and Robinson, H., Exploiting Smalltalk Modules In A Customizable Programming Environment, *Proceedings of ICSE 21, International Conference on Software Engineering*, Los Angeles, May 1999.

[8] Goldberg, A. and Ross, J., Is the Smalltalk-80 System for Children?, *Byte*, 6(8), August 1981.

[9] Holland S., Griffiths R., Woodman M., Avoiding Object Misconceptions *Proceedings SIGCSE 97*, San Jose, February 1997

[10] Krasner G. E., Pope S. T., *A Cookbook for using the Model View Controller User Interface Paradigm* in Smalltalk 80 Journal of Object-oriented Programming, Vol 1, #3 pages 26–49, 1988.

[11] Howard, T., *The Smalltalk Developer’s Guide to VisualWorks*, SIGS Books, New York, 1995.

[12] Gamma E., Helm R., Johnson R., Vliffides J. *Design Patterns: Elements of re-usable Object-oriented Software*, Addison Wesley, 1994.
[13] Alpert S. R., *The Design Patterns Smalltalk Companion*, 1998, Addison Wesley, 1998.